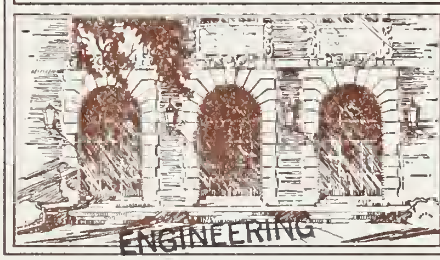


Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document No. 125

THE CARE AND FEEDING OF THE PEESPOL COMPILER

by

David M. Grothe

August 15, 1974

CAC Document No. 125

THE CARE AND FEEDING OF THE PEESPOL COMPILER

by

David M. Grothe

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

August 15, 1974

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the U.S. Army Research Office under Contract No. DAHCO4-72-C-0001.

ABSTRACT

The purpose here is to set forth the general strategies of implementation which are difficult to discern from reading the program itself. Detail is assiduously not paid attention to here. It is assumed that equipped with the knowledge of the general intent of things one can sooner or later discover all the detail that is necessary by consulting a listing of the compiler itself.

THE CARE AND FEEDING OF
THE PEESPOL COMPILER

0. INTRODUCTION

"The care and feeding of the PEESPOL Compiler" is a document intended for those who are stout of heart, firm in their resolve, not the author, and find themselves in the unenviable position of having to augment, change, modify, recompile, or otherwise approach the PEESPOL compiler as "just another Algol program."

The purpose here is to set forth the general strategies of implementation which are difficult to discern from reading the program itself. Detail is assiduously not paid attention to here. It is assumed that equipped with the knowledge of the general intent of things one can sooner or later discover all the detail that is necessary by consulting a listing of the compiler itself.

This document is not addressed to the casual reader, although those who are possessed by a bizarre curiosity are invited to peruse it until either boredom or confusion overwhelms them; it is rather recommended that one read this out of necessity.

Assuming that necessity is the motivating force, one should have a working familiarity with Burroughs B6700 extended Algol and an intimate understanding of the PEESPOL language and the PDP-11 instruction set.

1. FILES THAT COMPRISE THE PEESPOL COMPILER:

* The Compiler Proper:

PEESPOL/LIST INCLUDES	
PEESPOL/NO LIST INCLUDES	INCLUDES
PEESPOL/INCLUDES	
/GDECS	GLOBAL DECLARATIONS
/OPENBRACKETS	
/CLOSEBRACKETS	
/ELBATDEFINES	
/XREF/DEFINES	
/NES/EMITTERS/GDECS	
/TEXTGLOBALS	
/BINDDECS	
/DEFINE/DECLS	
/B6500	
/NEW/REGDECS	
/XREF/GDECS	
/FORWRD	FORWARD PROCEDURE DECLARATIONS
/DDECS	
/TEXTDECS	
/METAOP	
/ANTSPRINT	
/XPAND	
/XREF/PROCS	
/INFOPRINT	

PEESPOL/SCANNER

/ROMANNUMERALS

/MYCREATEDATE

/TBLSTUF

/TBLFILE625

/SERVIS

/LINKEDLIST

/NEW/REGISTERS

/NEW/REGSAVERS

/NEW/EMITTERS/0

/NEW/EMITTERS/1

/NEW/EMITTERS/2

/BRANCH

/BINDER

/CTE

/AEXP

PEESPOL/CONDEXP

/DISASSEMBLE

/GCOMP

/DECLS

/STMTS____

/FIRSTX FIRST EXECUTABLE CODE

* Compile Decks

PEESPOL/COMPILEDECK

Compiles PEESPOL/HOST/PEESPOL without a listing. This is a complete compile.

PEESPOL/COMPILEANDLIST

Complete compilation of PEESPOL/HOST/PEESPOL with a listing.

*PEESPOL/INFO

This is a file created by the Algol compiler in the course of either of the above two compilations. It is a saved copy of the Algol compiler's symbol table and is necessary in order to do separate compilations of the compiler.

*PEESPOL/LOG

This is a card image file into which the compiler logs runs of itself. This file is not necessary for the operation of the compiler. If it is not present, no logging takes place. The file must be created with Cande or some other editor that allocates a large-ish amount of disk space for a file.

*PEESPOL/XREFANALYZER

This is the code file for the cross reference analyzer.

*PEESPOL/XREF/ANALYZER

Source for the cross reference analyzer.

*EXPANDER

Code for the asynchronous expander.

*PEESPOL/EXPANDER

Source for the expander.

*PEESPOL/ELCLASSES

A file that the XREFANALYZER needs to run. A card image file.

*PEESPOL/INITIALINFO

This is the source for the initial state of the compiler's symbol table. The compiler itself processes this file and produces a file called PDP11/FILLS which contains the declaration of an Algol procedure called INITIALIZEINFO. This file's name must be changed to PEESPOL/TBLFIL625 in order to have it compiled into the compiler.

*PEESPOL/HOST/PEESPOL,PEESPOL/X/PEESPOL,
PEESPOL/NEW/PEESPOL

Various names for the compiler listed in increasing order of state of debuggedness

*XPEESPOL

WFL deck that copies PEESPOL/HOST/PEESPOL to PEESPOL/X/PEESPOL.

*NEWPEESPOL

WFL deck that copies PEESPOL/X/PEESPOL to PEESPOL/NEW/PEESPOL.

*PEESPOL/BUGS/=

A family of files which contain documentation for fixed bugs in the compiler. PEESPOL/BUGS/CURRENT tends to document bugs fixed in PEESPOL/X/PEESPOL, and the others tend to document bugs fixed in PEESPOL/NEW/PEESPOL.

*PEESPOL/ARCHIVEA, PEESPOL/ARCHIVEB

Files to accomplish dumping of these files to tape.

*PEESPOL/REMOVER

File to accomplish the removal of all PEESPOL files except those that are necessary for the operation of the compiler.

*PEESPOL/COMPILE/PEESPOL

Source for a program (OBJECT/COMPILE/PEESPOL) which aids in doing separate compilations of the PEESPOL compiler.

2. HOW TO COMPILE THE COMPILER

There are two files which, when scheduled for execution, will compile the PEESPOL compiler. These are:

PEESPOL/COMPILEDECK

PEESPOL/COMPILEANDLIST

The result of either compiler will be a program called PEESPOL/HOST/PEESPOL, and a dump of the Algol compiler's symbol table in a file called PEESPOL/INFO.

The first compile deck does not produce a listing and the second one does.

Note that the files which constitute the compiler itself must be present on disk.

3. HOW TO DO SEPARATE COMPILES

With the aid of an interactive program, it is straightforward to modify one of the files which comprise the compiler and to compile only the procedures in that file, automatically binding them into the compiler.

The following files must be present to accomplish this:

OBJECT/COMPILE/PEESPOL	- The Program
PEESPOL/HOST/PEESPOL	- Host for the Bind
PEESPOL/INFO	- Algol's symbol table
+ Those files which were modified	

To run the program type:

E COMPILE/PEESPOL

The program will then request certain information:

DESTINATION:

Give a valid argument for the TO= construct in WFL.

NAME FOR THE COMPILE:

Give a valid argument for the NAME= construct in WFL or simply a carriage return.

AUTOBIND?

Give a YES or NO answer, CR means yes.

The program will then ask for the names of the files to be compiled. It will prompt with the text "INCLUDE PEESPOL/". After all the files names that are to be compiled and bound are entered, type an empty line to terminate file name entries.

The program will then ask:

GO?

Respond with YES or NO, CR means yes.

Example (program output is underlined):

E COMPILE/PEESPOL

DESTINATION: RJE <cr>

NAME FOR COMPILE: SEPARATE/DECLS <cr>

AUTOBIND? <cr>

INHIBIT LISTING? <cr>

INCLUDE PEESPOL/ DECLS <cr>

INCLUDE PEESPOL/ <cr>

GO? <cr>

JOB # IS mmmm

4. THE STRUCTURE OF PEESPOL AS A PROGRAM

The three main syntactic sections of the PEESPOL compiler consist of global storage and define declarations, procedure declarations, and the main executable code.

Most of the procedures have either no parameters or a short parameter list. The style of implementation is to code procedures that are analogous to machine instructions in that they act on a certain predetermined set of data.

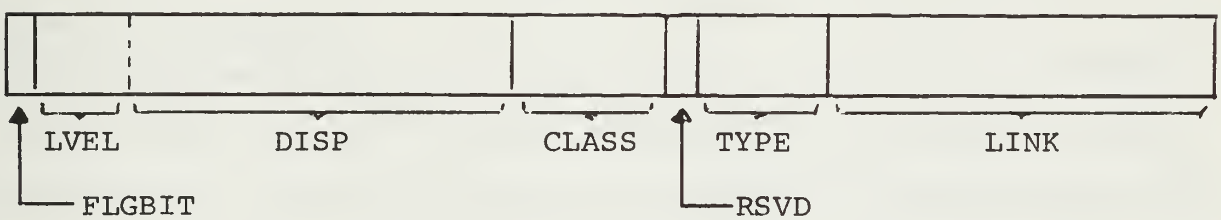
5. GENERAL STRATEGIES OF SUBSYSTEMS IN THE COMPILER

This section gives a general overview of various subsystems in the PEESPOL compiler. Not much attention is paid to detail; the curious can consult a listing of the compiler itself for details. The intent here is to set forth the general strategies so that one can approach the listing of the compiler with an idea of what process a given section of code purports to implement.

5.1 SCANNER

The general architecture of the scanner is that the parsers of the compiler be isolated as much as possible from the raw input to the compiler. The interface between the scanner and the parser is queue of one-word LEXEMES called ELBAT (table spelled backwards, after the name of the procedure which maintains the queue). Frequent mention will be made to "ELBAT WORDS;" these are words of the format of the words found in the ELBAT queue.

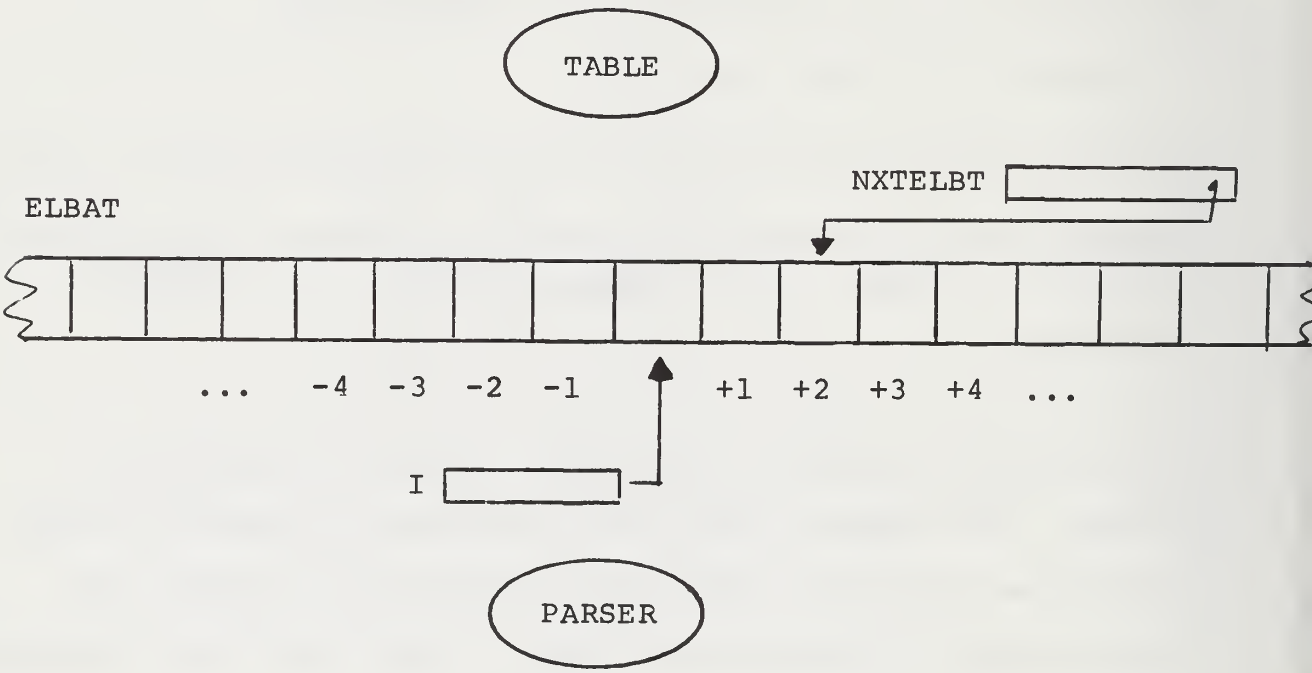
Format of an ELBAT word:



This word can describe a token sufficiently for purposes of parsing. The address field gives the machine address assigned to the time, with the LVEL field distinguishing amongst the various kinds of addresses that exist. The class field designates

the syntactic class of the item, for example whether it is a word identifier, byte procedure identifier, the word "IF", etc. The type field is a sort of sub-class field which is used to distinguish such things as procedure declared forward, call by reference parameter, "DOPED" array, etc. The link field contains an index into the symbol table to the entry corresponding to this ELBAT word.

A global integer in the compiler, "I", denotes the current word of the ELBAT queue. A procedure called TABLE accepts an index into ELBAT and returns the class field of the corresponding ELBAT word. In addition, TABLE knows which ELBAT words are valid and which ones are not, so tokens may be extracted from the input stream in order to validate the requested ELBAT word. The following illustration will clarify this:



In this illustration a call on table with argument I+2 will result in a scan of the input stream since ELBAT [I+2] is not yet valid.

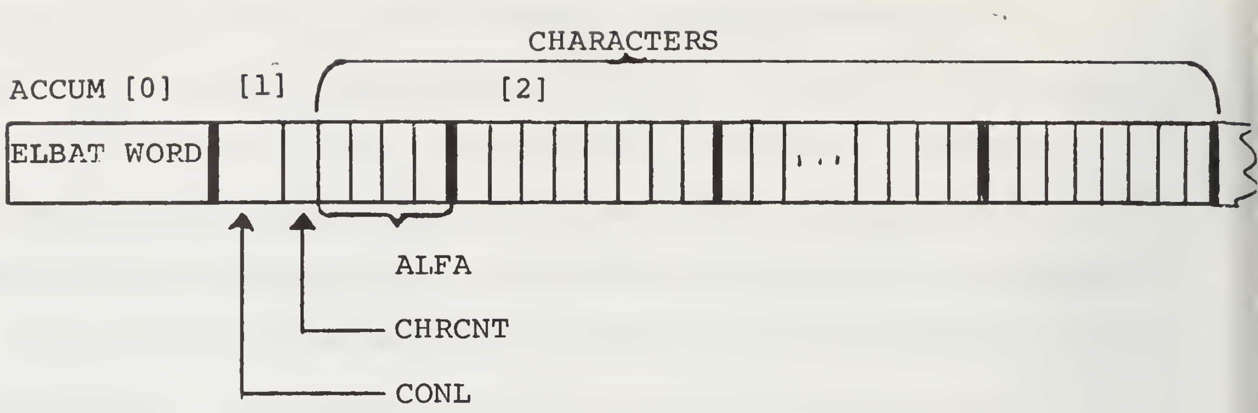
The agreement with TABLE is that if it is called with argument P, upon return ELBAT [P] will contain a valid LEXEME. The argument is constrained to be relative to "I".

TABLE has the license to initiate arbitrary computations in order to validate ELBAT [I±8].

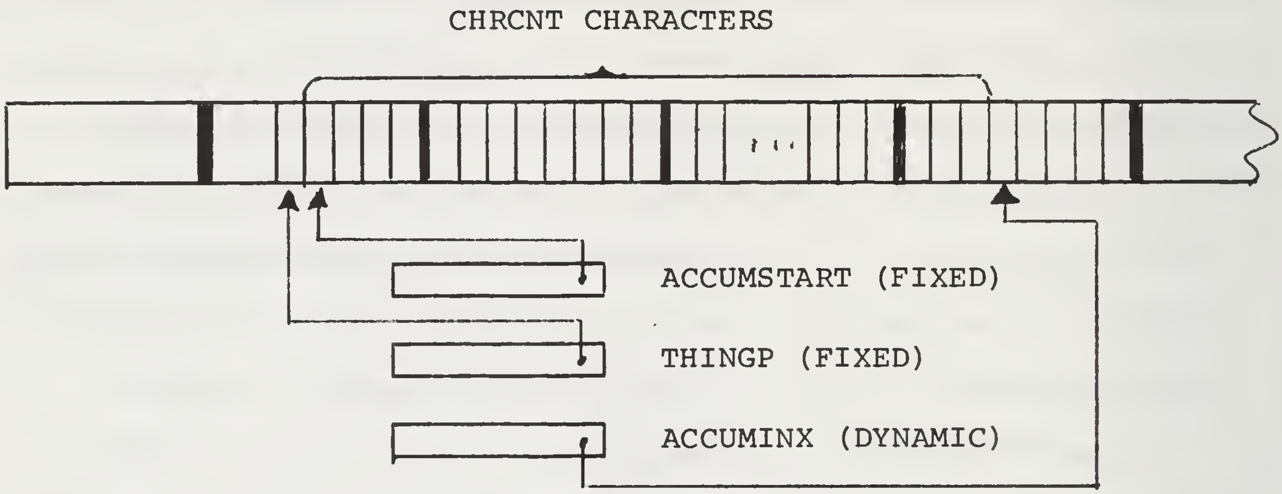
One of the kinds of computation that may occur is to invoke a define. Define invocations take place in TABLE at the time that a symbol has been scanned from the input stream and looked up in the symbol table, but <u>before</u> LEXEME is inserted into ELBAT.

Another kind of computation that may occur is the invocation of a metafunction. A metafunction is so-named because it <u>tells</u> TABLE upon return what kind of a LEXEME to insert into ELBAT (or possibly none at all). A metafunction computes a LEXEME.

When TABLE finds it necessary to scan the input stream, it invokes a procedure called SKAN. SKAN is driven by some global input stream pointers and deposits the text of the next token in the input stream in a global charactor accumulator. This character accumulator has the same format as a symbol table entry. Here is a partial illustration of its contents:



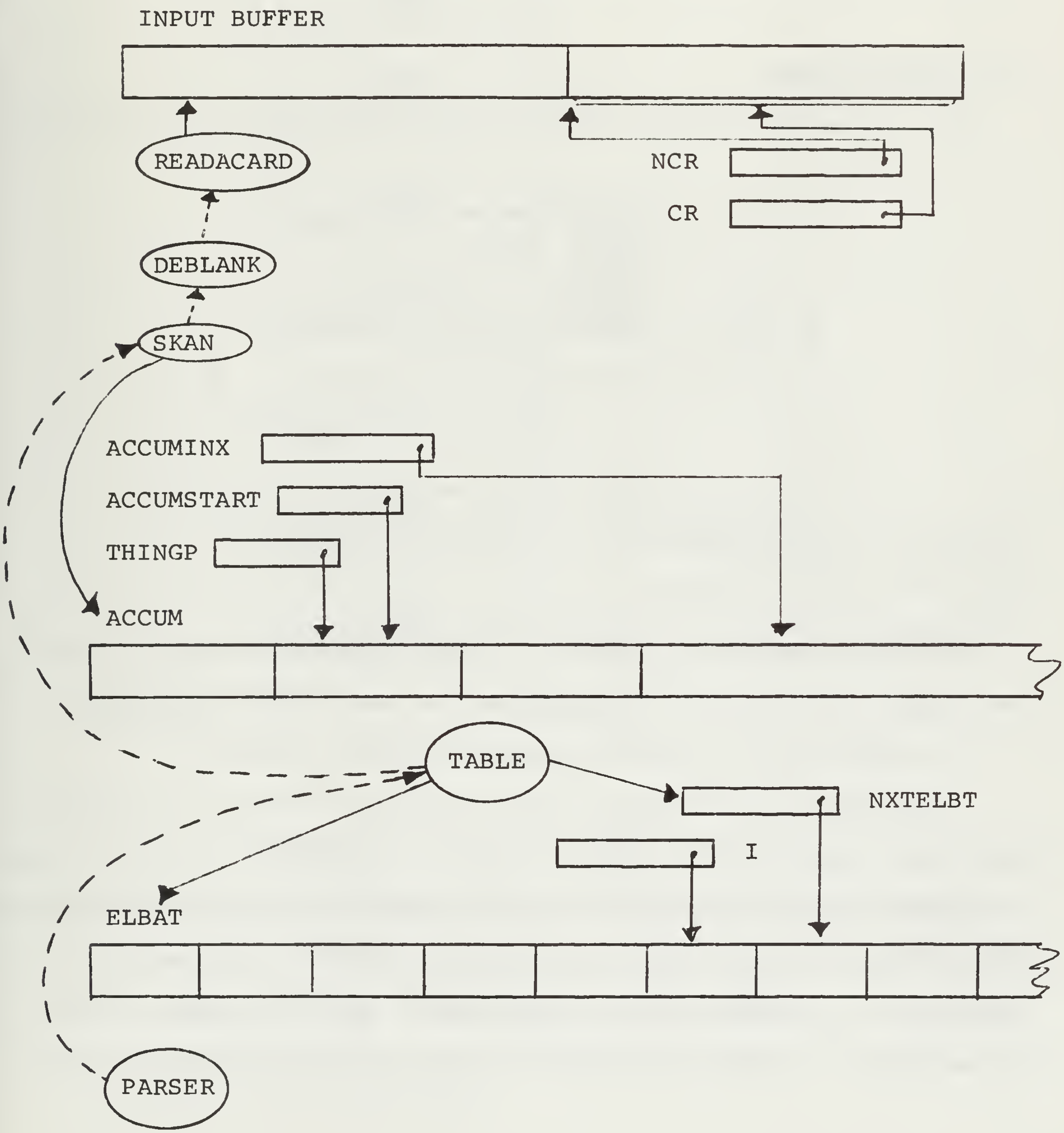
Certain pointers designate locations in ACCUM:



The input stream that is processed by SKAN is defined by a next-character pointer (NCR) and an integer giving the number of characters remaining in the current input buffer (CR).

When the end of the current input buffer is reached, a procedure called "READACARD" is invoked to get another card-image from the input. READACARD is invoked as a consequence of DEBLANKING to the end of this current input buffer. The following

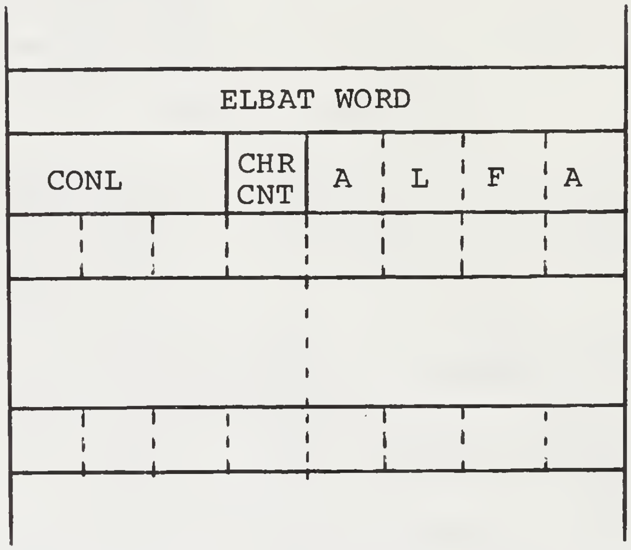
is a schematic diagram of the scanner. Procedures are enclosed in "circles;" solid arrows emanating from a circle denote the things that are modified by the procedure; dashed arrows indicate calling chain; data and pointers are enclosed in rectangles.



Two intermediate procedures are often used to call TABLE. These are called "STEPI" and "STEPIT." They each call TABLE with argument I:=I+1 and store the result returned in a global called "ELCLASS." In this manner, there is usually a correspondence maintained between the value of ELCLASS and ELBAT [I].CLASS.

5.2 SYMBOL TABLE

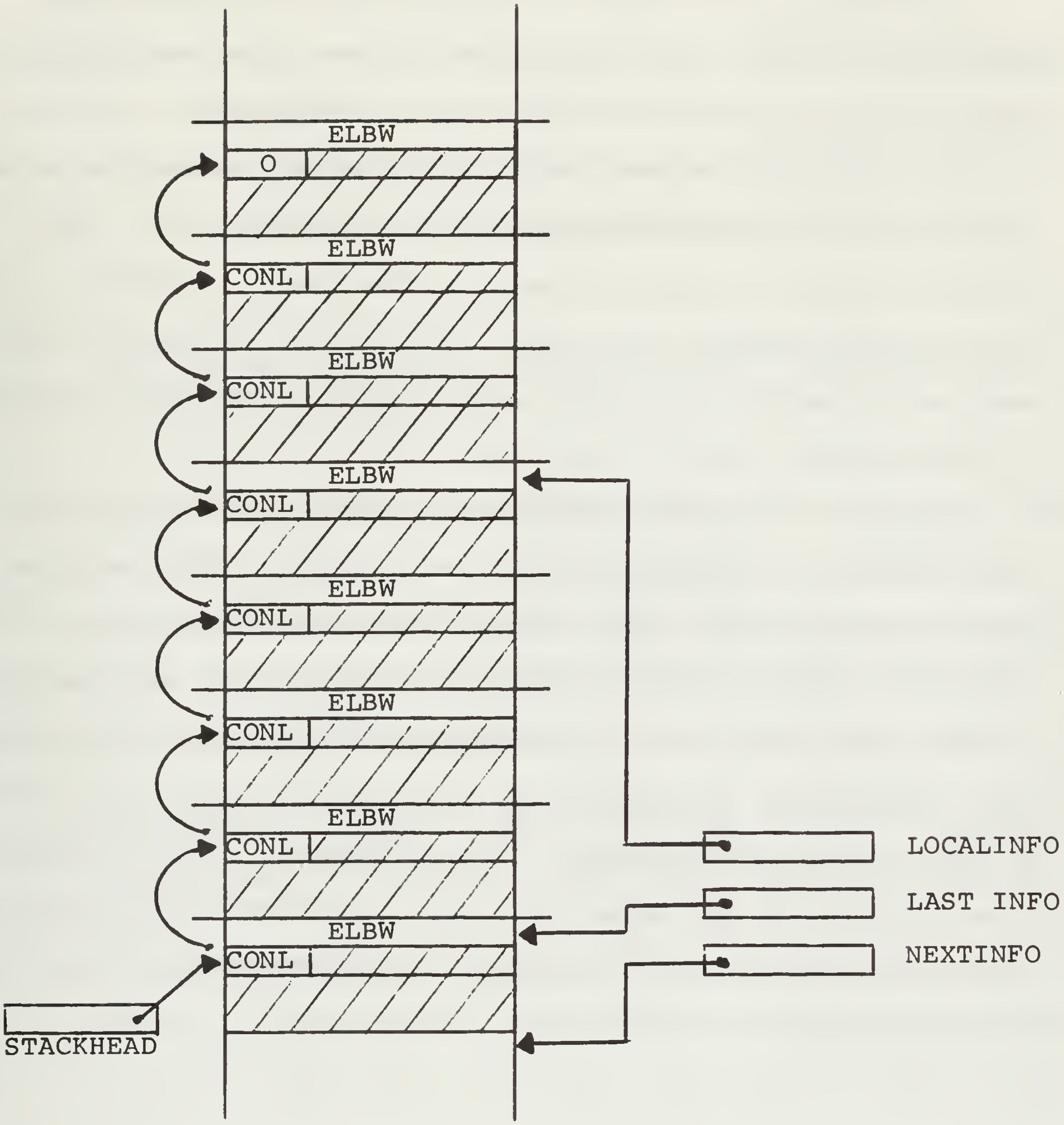
The format of a symbol table entry (INFO) is illustrated here:



The link field of a LEXEME in ELBAT indexes the ELBAT word of the symbol table entry corresponding to that LEXEME.

The CONL field is used to thread entries together.

A simplified view of the macro structure is that new entries are added onto the end of the table with the CONL field of the new entry pointing to the word containing the CONL field of the previous (last) entry in the table, with a global word in the compiler pointing to the last entry in the table. Thus the table has an organization of LIFO queue with the CONL field carrying the link.



As can be observed from the above illustration, the organization of the table lends itself to block structured languages in that a local entry for a particular symbol will be found before a global entry for the same symbol. If one keeps a local/global marker (LOCALINFO) one can easily distinguish a local vs. a global occurrance of an identifier.

At block exit the table is "purged" by walking the CONL thread down to the first entry with an index less than LOCALINFO and resetting stackhead and NEXTINFO appropriately.

The difference between this simplified scheme and the actual organization is that there are really 625 stack heads. The lookup algorithm consists of hashing the symbol in ACCUM to produce an index between 0 and 624, using the corresponding stack head and proceeding as in the above diagram. The table consists of 625 threaded lists in one array.

There is a companion table, called ADDL, which is used to keep additional information for certain kinds of info entries. For these entreis the link field of the ELBAT word portion of the entry will point to the corresponding ADDL entry. An example of an INFO entry which has a corresponding ADDL entry is the entry for a procedure. The ADDL entry consists, basically, of a list of ELBAT words corresponding to the parameters of the procedure. These ELBAT words are consulted when a procedure invocation is being compiled in order to enforce actual/formal parameter type correspondence.

5.3 DEFINES

The general strategy for the implementation of defines goes something like this:

- Save the define text somewhere and save a pointer to it in the address field of the INFO entry for the define name.

- Put the parameter names into the symbol table, but mark them as "undeclared."

- Associate a list of "parameter descriptors" (kept in ADDL) with the entry for the define in INFO. A parameter descriptor will point to the INFO entry for the parameter name and say something about what it is that terminates the actual parameter text.

- When TABLE looks up a define, consult the list of parameter descriptors and save the actual parameter text somewhere.

- Give the parameter names in INFO a class of define parameter and point each of their address fields to the actual parameter text that was saved. Save the old ELBAT words on a stack.

- Save the input stream pointers on a stack and point NCR at the define text.

- Go back to the top of TABLE and call SKAN again

- When table looks up a define parameter, push the input pointers and point them to the associated parameter text.

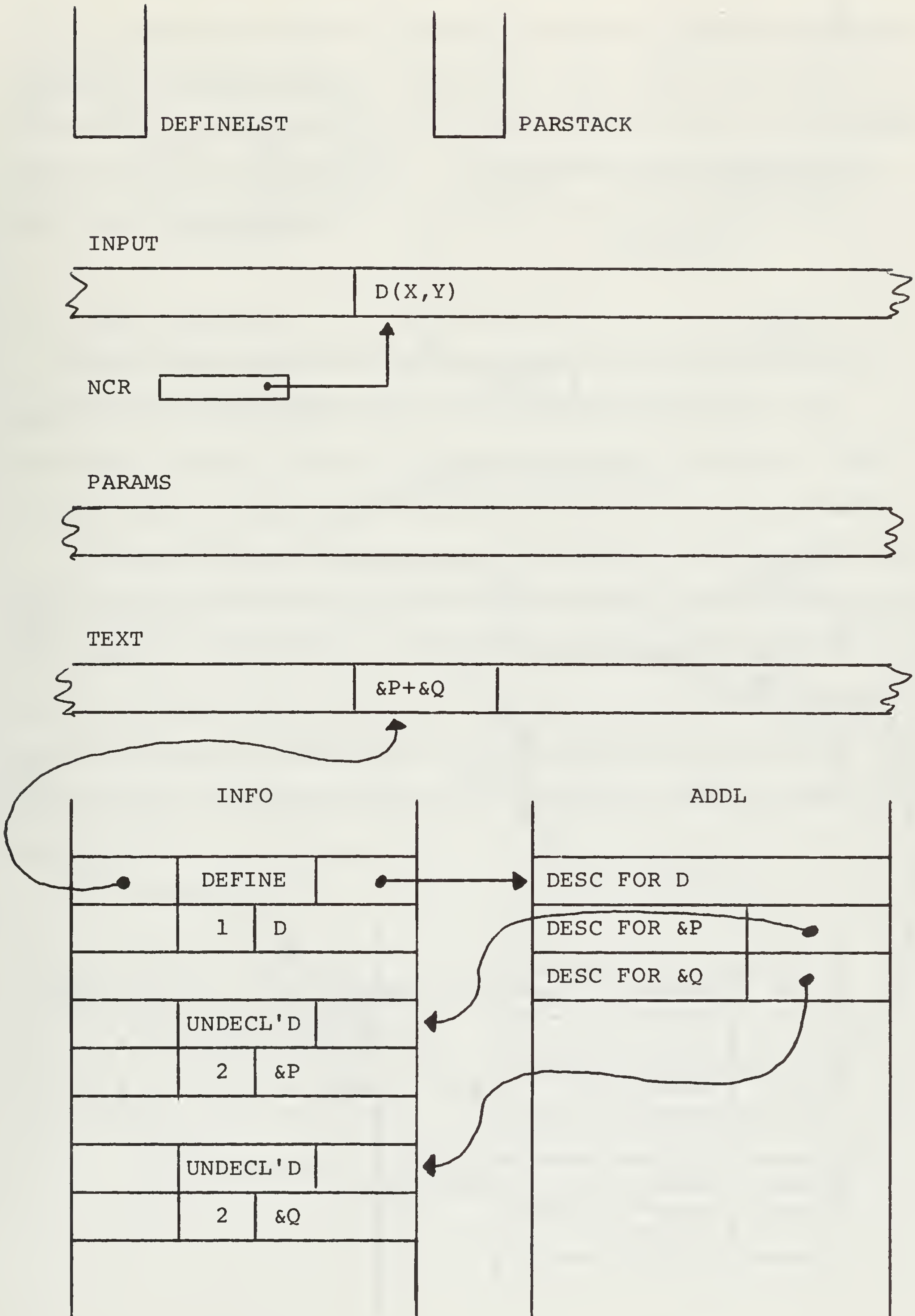
- When an "END OF TEXT" marker is encountered in the course of scanning define or parameter text, pop the input pointers and proceed.

The next two illustrations depict the state of the various tables "before" and "after" the invocation of a define. As a working example we will assume that we have the following define declaration:

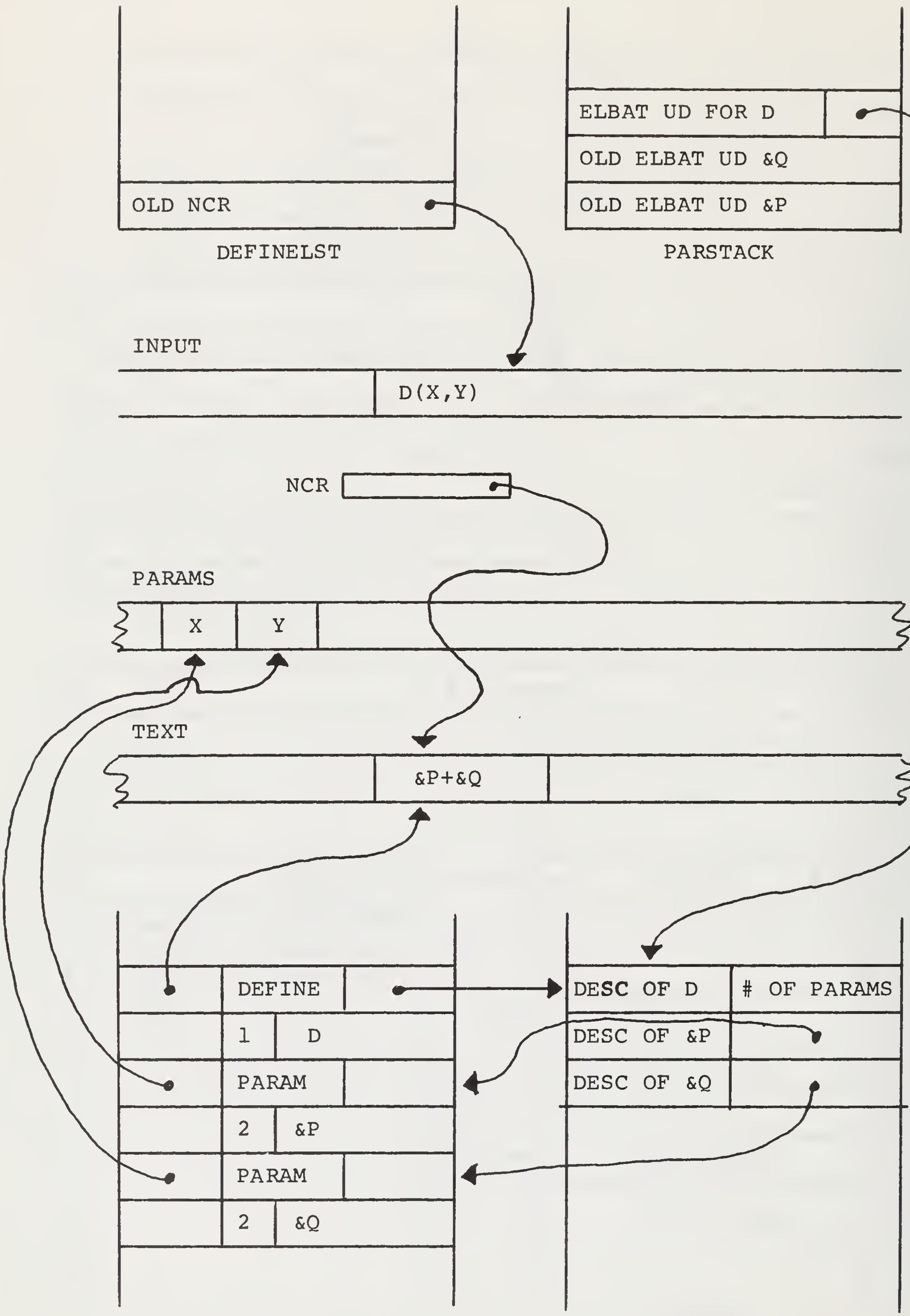
```
DEFINE D(&P,&Q) = &P + &Q ##;
```

and the following define invocation:

```
D(X,Y)
```


DEFINELST
PARSTACK
INPUT
D(X,Y)
NCR
PARAMS
TEXT
&P+&Q
INFO
ADDL
DEFINE
1
D
UNDECL'D
2
&P
UNDECL'D
2
&Q
DESC FOR D
DESC FOR &P
DESC FOR &Q

BEFORE


ELBAT UD FOR D
OLD ELBAT UD &Q
OLD ELBAT UD &P
OLD NCR
DEFINELST
PARSTACK
INPUT
D(X,Y)
NCR
PARAMS
X
Y
TEXT
&P+&Q
DEFINE
1
D
PARAM
2
&P
PARAM
2
&Q
DESC OF D
OF PARAMS
DESC OF &P
DESC OF &Q

AFTER

The process of exiting a define begins by popping the ELBAT word for the define off of PARSTACK. From there the number of parameters it has can be found from its ADDL entry and the ELBAT words restored from PARSTACK. The input stream pointers are restored from DEFINELST.

5.4 META FUNCTIONS

When TABLE looks up a Meta Function it calls a procedure, passing it a case code, which performs the function. This procedure returns a class field setting which instructs TABLE as to what kind of LEXEME to build or as to what action to take.

Any computation whatever can be performed, but special care must be taken if the computation must invoke the parser procedures, for example to evaluate a compile-time expression. The difficulty arises because the parsers will recursively call TABLE, causing ELBAT to change. Therefore, any Meta Function which needs to use ELBAT, first saves it locally and restores it once the Meta Function is complete.

5.5 CODE EMITTERS

PEESPOL maintains three memory images: save memory, overlay code, and segmented data. These memory images are built such that increasing B6700 memory addresses correspond to increasing PDP11 memory addresses:

B6700

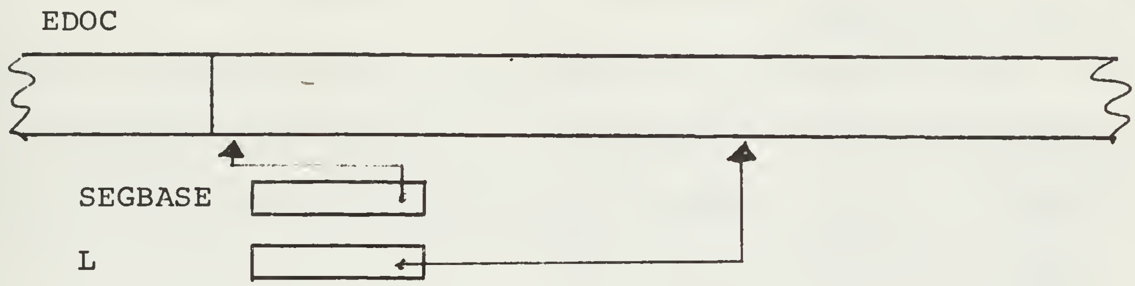
0	1	2	3	4	5
6	7	8	9	10	11

PDP11

1	0
3	2
5	4
7	6
9	8
11	10

Code for any executable program unit is queued up in the same format in a buffer apart from these memory images and, when the "segement" of code is complete, moved into the appropriate

memory image. The code buffer is called "EDOC" (code spelled backwards). A global integer, "L," indicates the next byte into which code will be stored, and a global integer, "SEGBASE," indicates the EDOC location of the first byte of code for the segment currently being compiled.



Code is put into EDOC by calling one of a family of procedures which constitutes the code emitters.

The code emitters are organized in a hierarchy which is three levels deep.

At the lowest level are the basic emitters which put a word of code or data into EDOC.

The next level contains emitters which handle the various families of PDP11 instructions, such as single and double operand instructions.

The highest level contains specialized emitters which interact more strongly with the parsers. These emitters do things like accomplish an arithmetic operation given an ELBAT word for the operator.

The branch emitter is given a branch instruction and a source and destination "L" value. It generates whatever code is necessary to cause control to transfer from one specified location to another.

The emitters put the code wherever "L" points. Code can be generated out of order, say for forward branch fixups, by saving L, planting a place holder, and coming back to that location later.

5.6 OPTIMIZING

Code optimization is done using the "peephole" technique. The optimization is done from within the code emitters themselves; thus the parsers generate canonical form code and the emitters do localized compression on it.

One of the optimization strategies has to do with removing extraneous MOV instructions. An example will illustrate the technique:

```
A:=B+C;
  Canonical form code:
        MOV B,R4
        MOV C,R3
        ADD R3,R4
        MOV R4,A
Optimized code:
        MOV B,R4
        ADD C,R4
        MOV R4,A
```

This heuristic is implemented at the point of emitting a two address instruction. The emitter, when told to emit ADD R3,R4, takes note of the fact that the previous instruction was a MOV

instruction and that its destination (R3) was the same as the source of the ADD instruction. When this situation is recognized, "L" is set back to the location of the MOV instruction, and the MOV gets overwritten with the ADD. (and the remark "oops..." is printed on the code listing).

To aid in the implementation of the above scheme, the emitters maintain a global, "LASTMOV," which is the "L" value of the last MOV instruction if and only if the last instruction generated was a MOV instruction, and is otherwise set equal to -1. Associated with that are two words, "LASTMOVSRC" and "LASTMOVDST", which are encodings of the source and destination of the last MOV instruction.

Another optimizing strategy concerns the optimizing of CMP instructions. An example illustrates the technique:

```
        A EQL B
    Canonical form code
        MOV A,R4
        MOV B,R3
        CMP R3,R4
    Optimized code:
        CMP B,A
```

The heuristic employed is to recognize the two consecutive MOV instructions immediately preceding the CMP instruction and to overwrite them with the CMP instruction.

Another optimizing heuristic exists for the :=* operator. Example:

```
A:=*+B
```

Canonical form code:

```
MOV A,R4
ADD B,R4
MOV R4,A
```

Optimized code:

```
ADD B,A
```

The heuristic is employed at the point where the instruction MOV R4,A would have been generated and consists of observing that the only thing that happened to A after it was moved to R4 was that B was added to it.

In order to aid the :=* optimization, the emitters record the most recently generated instruction and its corresponding source and destination in some global variables. These are:

LASTOPL	"L" of last instruction
LASTOPINST	Last instruction
LASTOPSRC	Source of last instruction
LASTOPDST	Destination of last instruction
LASTOPVALID	True if this information is valid, i.e., the last instruction wasn't a branch or something.
LASTOPDBL	True if the last instruction was a two address instruction.

5.7 PARSING TECHNIQUE

The parser is a hand-coded, recursive descent parser. It resembles a top-down parsing technique except that no blind alley is ever taken, hence there is no back-tracking. Most of the language can be parsed with no lookahead.

The file PEESPOL/ELBATDEFINES contains a list of defines that mnemonically identify the values of the class field in an ELBAT word. To illustrate how this parsing technique is used the following procedure is offered as an example of a recognizer for an "IF STATEMENT":

```
PROCEDURE IFSTMT;
BEGIN
   IF ELCLASS ISNT IFV THEN
      ERR ("IF EXPECTED");
   STEPIT;
   CONDITIONALEXP;
   IF ELCLASS ISNT THENV THEN
      ERR ("THEN EXPECTED")
   ELSE STEPIT;
   STATEMENTLIST;
   IF ELCLASS IS ELSEV THEN
   BEGIN
      STEPIT;
      STATEMENTLIST;
   END;
   IF ELCLASS ISNT FIV THEN
      ERR ("FI EXPECTED")
```

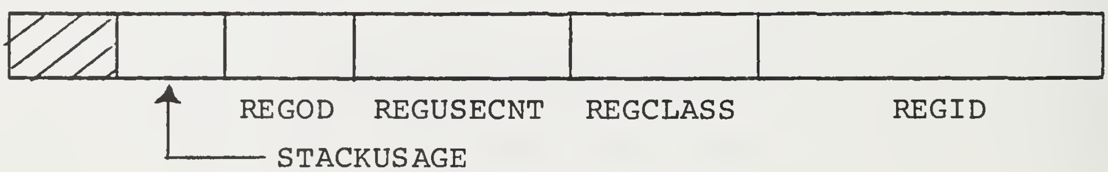
```
        ELSE
            STEPIT;
END         IFSTMT;
```

In the compiler itself, code is generated as the parsing proceeds. The code generation is based on a stack model of the PDP11 in which the general registers are regarded as the top of the operand stack, with overflow into the memory stack. Thus, the code for an arithmetic primary consists of moving a value to a register and designating that register the top-of-stack. Arithmetic operations are performed on the two logical top-of-stack registers, etc.

5.8 REGISTER ALLOCATORS

The register allocators keep track of register usage and maintain the image of the logical operand stack. The allocators are written in a more general way than their actual use by the compiler would warrant. For example, the register allocators have the capability of keeping track of what is actually in the registers, but the compiler cannot exploit this capability since it does not distinguish so-called "basic blocks" (roughly, branch-less sequences of code).

A master table of the registers records the state of their usage. An entry in this table has the following format:



This word is called a "register control word" and is often abbreviated RCW in the compiler.

The logical operand stack (REGSTACK) is a stack of RCW's. A procedure exists to push the stack, which consists of obtaining a register and pushing its RCW onto the REGSTACK. Popping the stack amounts to deleting its RCW from the REGSTACK.

The REGUSECNT field of the RCW records the number of times the register was stored into. The value of this field is used at the end of compilation of a procedure or routine to determine which registers were used and must be saved and restored.

The STACKUSAGE field keeps track of the number of outstanding occurrances of this register in the logical operand stack.

The REGCLASS and REGID fields classify the register as to its usage and contents. Currently, only three states exist: Available, Contents Unknown, and Reserved User Register.

5.9 PREVIOUS LEVEL

The previous level facility in the compiler amounts to a checkpoint/restart capability. Enough information is stored in a PEESPOL code file that the compiler can resume compiling from the point at which the code file was created.

Internally the process is one of re-establishing the memory images and the symbol table from their saved state in the code file. Additionally, there is a vector called COMPILERSTATE which is used to save various parameters of the compilation--memory allocation pointers, for example. There is a master table in GDECS which defines which word means what in this vector.

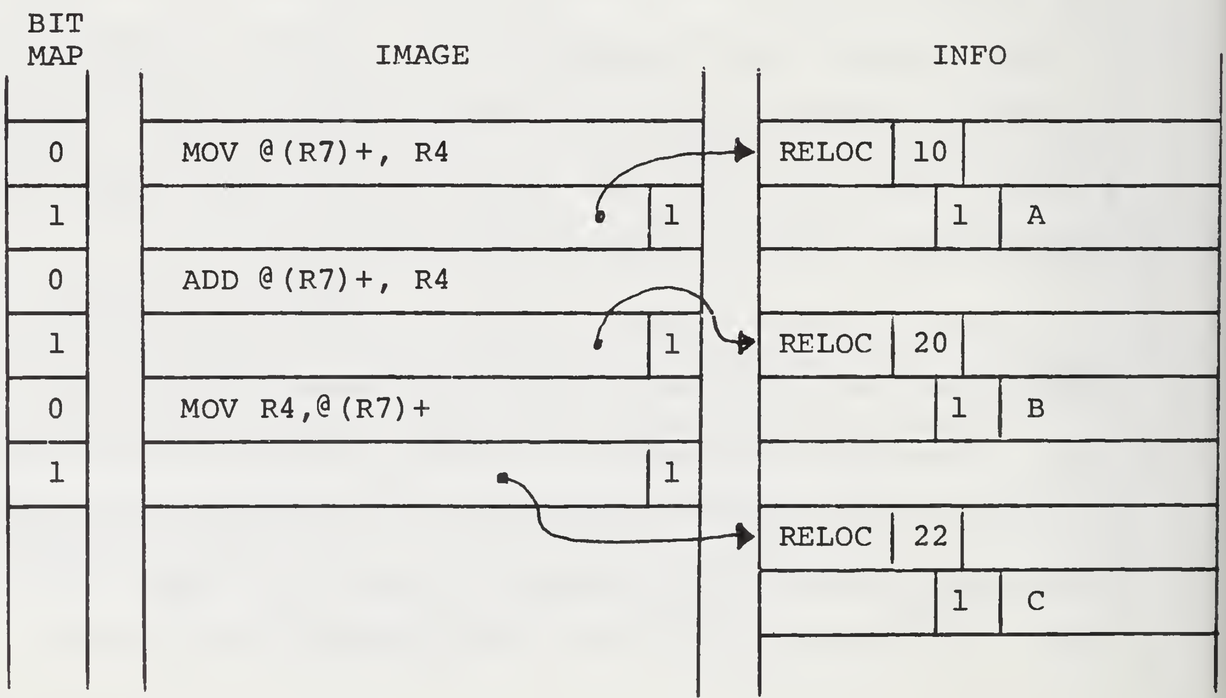
When bringing in the symbol table, one of two alternative techniques is used. If the symbol table from the previous level will fit onto the end of the compiler's symbol table, it is read directly into INFO. Otherwise, it is read into a temporary image of INFO and the entries are moved into INFO one at a time.

5.10 BINDING AND MODULE COMPILATION

The underlying data structure of a PEESPOL module is described in this section.

When compiling a module, all storage is allocated as relocatable. References to relocatable storage are noted through the use of bit maps. Every memory image-ish array in the PEESPOL compiler has an associated bit map. An on-bit indicates that the corresponding word in the image array contains a reference into INFO to the symbol table entry of the thing referred to.

Example:

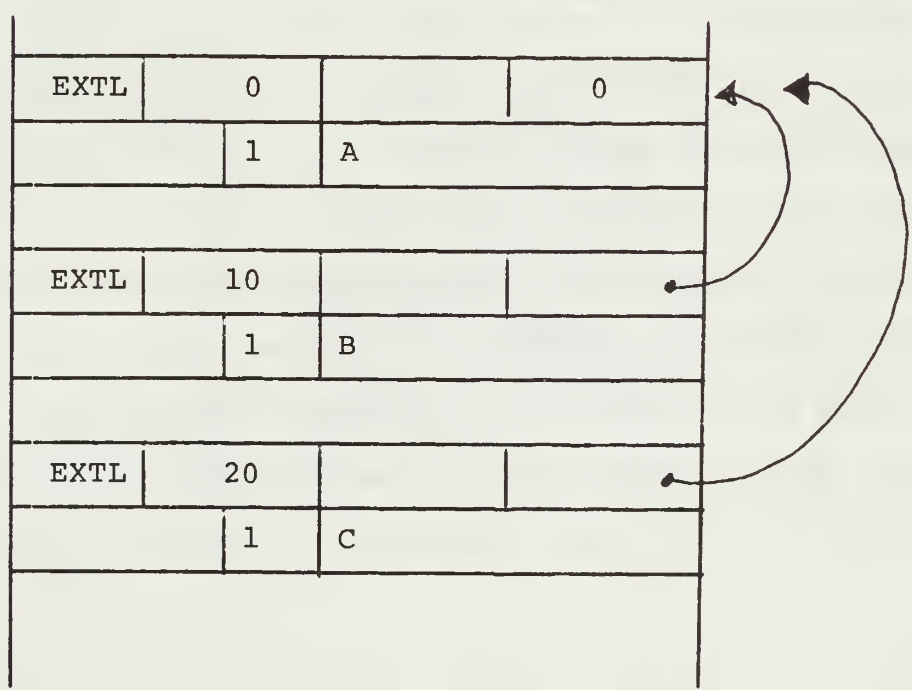


The above would be how the code for C:=A+B would look if A,B, and C were all relocatable. The LVEL field of the ELBAT word contains the indication that the address is relocatable. The 16-bit quantities in the image which point to INFO consist of a 15-bit index in the high order bits, with the low order BIT = 1. This is done to provide some redundancy checking at bind time.

The symbol table structure for externals is somewhat more complex. By way of address equation, it may happen that one external may depend on another. Example:

```
EXAMPLE WORD A;
WORD B = A+10K,
         C = B+10K;
```

If "A" were to be resolved to address 10, one would expect "B" to equal 20 and C to equal 30. In order to allow external addresses to depend on other external addresses, a linked structure is employed in INFO. The following illustrates the structure of the above example:



The linkage is implemented via the link field of the ELBAT word in the symbol table entry. Some symbol table entries have a non-zero link to ADDL (procedures, routines, arrays, and others). In the case of these entries, it is the link field of the ADDL word pointed to which carries any external dependence. Procedures within the compiler exist for fetching and storing into these link fields. These procedures select the INFO or ADDL link as appropriate.

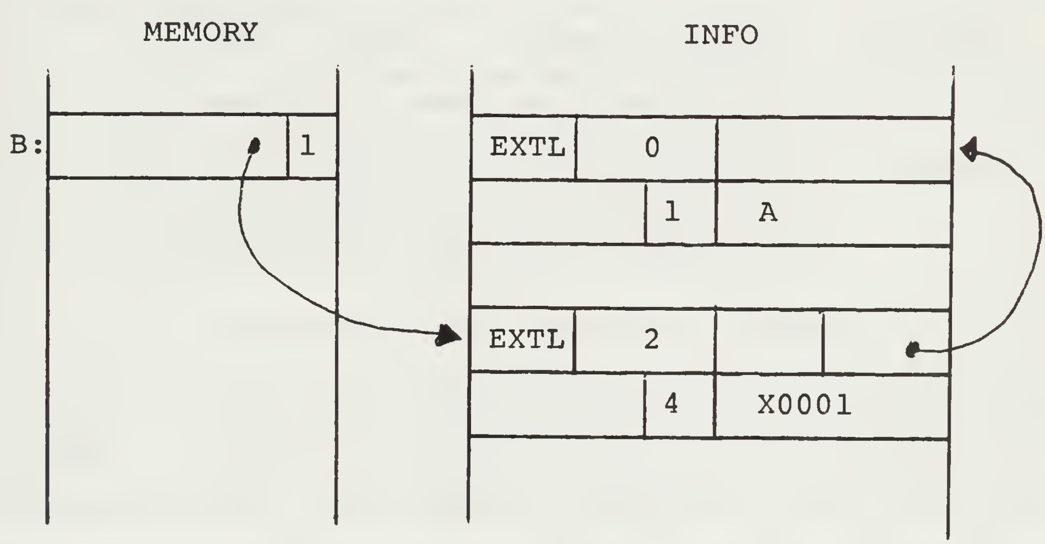
It is also a design constraint that the links by only one level deep. The compile-time-expression evaluators go to some trouble to enforce this. The valve returned by procedure CEXPRESS has the following format:

The BDISP and BLVEL fields are analogous to the DISP and LVEL fields of an ELBAT word; the only difference is that they are at the "Botton" of a word rather than at the "Top." The INFOF field is an INFO pointer which indicates the symbol upon which the expression result depends. If the LVEL field indicates that the value is external, the DISP field will be an offset relative to the external whose INFO index is contained in INFOF; for all other cases, the address (or baddress) field is the actual address as best it is known (i.e., possible relocatable). This return value from CEXPRESS facilitates establishing this kind of link structure in that the dependence is clearly displayed.

Sometimes an address is formed as a result of evaluating a C.T.E. which has no corresponding INFO entry. Consider the following section of code:

```
EXTERNAL WORD A;
WORD B: = A+2
```

What does the initial state of the word in memory corresponding to "B" look like? It must contain a pointer into INFO, but there is no declared symbol to which it can point. Under circumstances like this the compiler manufactures a symbol table entry purely for the purpose of having something to point to from the memory image. These entries are called "dummies" and are threaded together by a stackhead that does not correspond to any scramble modulus. These dummies can be either relocatable or external and are given symbolic names of the form Rmmm and Xmmm, respectively. The data structure that corresponds to the preceding example is:



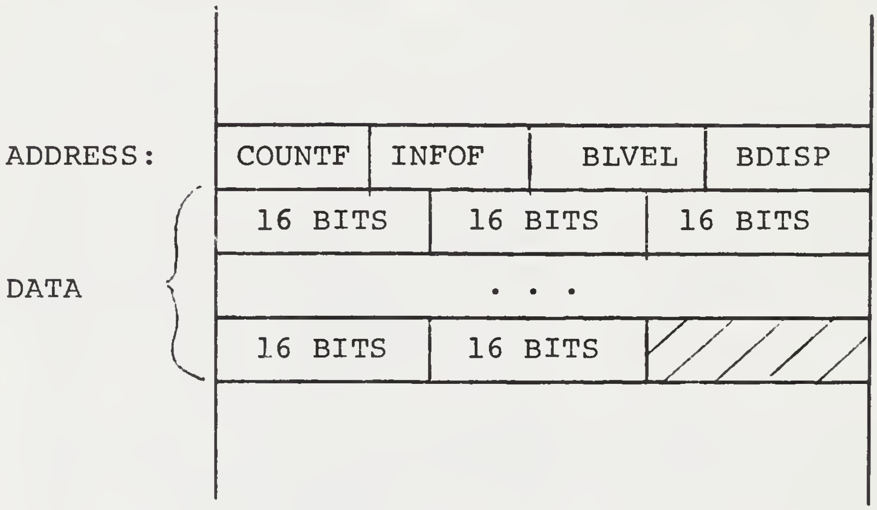
The dummy is an external whose linkage reflects the dependency.

It can also happen that a piece of storage can be equated to an external and initialized with some data. Consider the following section of code:

```
EXTERNAL WORD A;
WORD ARRAY B[*] = A: = 1,2;
```

In order to accomodate such things, PEESPOL implements what it calls "deferred memory." Deferred memory is simply a table of addresses and contents which cannot be placed in any of the memory images. The hope is that the location of the data will become known through the binding process and that the data will then be able to be placed at the correct address. The format of a deferred memory entry is:

DEFERREDMEM



Each entry is aligned on a B6700 word boundary. The first B6700 word is a C.T.E. formatted address with a byte count appended to the high-order bits. The remainder of the entry is a stream of

bytes as it is to appear in memory once its location becomes known. There is a bit map which corresponds to DEFERREDMEM in the usual way, i.e., successive bits of the bit map correspond to successive 16-bit chunks of the image.

The binding process consists of those steps that are necessary to resolve relocatable addresses that are found in the module being bound in and to resolve external references that occur in either the module being bound or in the program into which it is being bound. The steps are essentially these:

- Read in all the image arrays and their bit maps from the module's code file.

- Read in the module's symbol table to a temporary symbol table.

- Walk the module's symbol table relocating and attempting to resolve externals by looking them up in the program's symbol table.

- Fix up addresses in the images read in by scanning the bit maps and replacing symbol table pointers with actual addresses.

- Walk the program's symbol table and attempt to resolve any externals in it by looking them up in the module's symbol table.

- Walk the deferred memory table and place any resolved data areas where they belong, fixing addresses, and copy those which cannot be resolved onto the end of the program's own deferred memory table.

- Merge the module's symbol table into the program's symbol table.

At the end of every compilation, a binding step is performed to, as it were, bind the program to itself. This binding step is the same as the others except that, of course, no images are read in from anywhere and no symbol table merging takes place. It consists mainly of a bit-map scan and a walk of deferred memory to fill in addresses that were "forward referenced" in the binding process.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER CAC Document No. 125	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Care and Feeding of the PEESPOL Compiler		5. TYPE OF REPORT & PERIOD COVERED Research Report 6. PERFORMING ORG. REPORT NUMBER CAC-125
7. AUTHOR(s) David M. Grothe		8. CONTRACT OR GRANT NUMBER(s) DAHC04-72-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Advanced Computation University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 1899
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia		12. REPORT DATE August 15, 1974 13. NUMBER OF PAGES 40
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) U. S. Army Research office Duke Station Durham, North Carolina		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Copies may be requested from the address given in (9) above.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restriction on distribution

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Compiler
Block-structured languages
Computer languages
Meta-language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose here is to set forth the general strategies of implementation which are difficult to discern from reading the program itself. Detail is assiduously not paid attention to here. It is assumed that equipped with the knowledge of the general intent of things one can sooner or later discover all the detail that is necessary by consulting a listing of the compiler itself.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)